Как мы добавляли распределенный SQL в Hazelcast

Владимир Озеров
Querify Labs





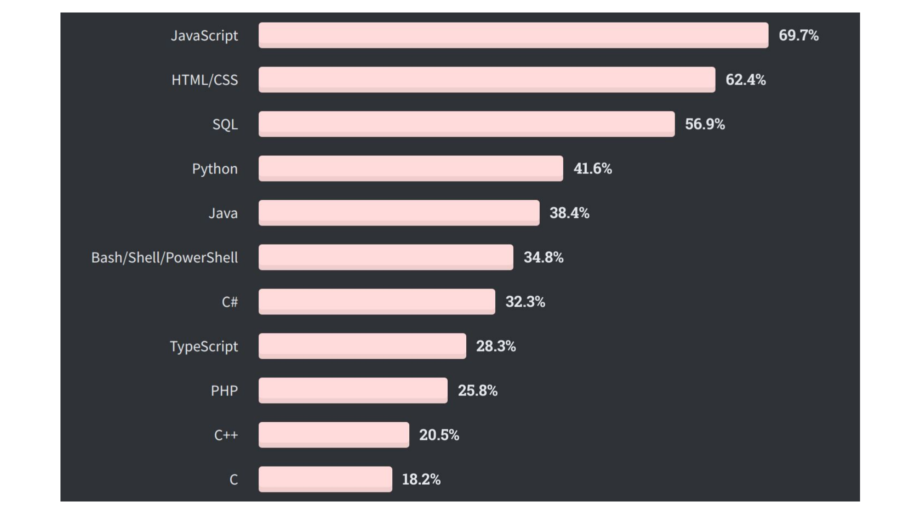
Кто использует SQL?

- “Классические” СУБД (MySQL, Postgres, SQL Server, Oracle)
- “Новые” системы управления данными
    - Реляционные (CockroachDB, TiDB, YugaByte)
    - BigData/Analytics (Hive, Snowflake, Dremio, Clickhouse, Presto)
    - NoSQL (DataStax*, Couchbase*)
    - Compute/streaming (Spark, ksqlDB, Apache Flink)
    - In-memory (Apache Ignite, Gigaspaces)
- Бунтари
    - MongoDB
    - Redis

* Используют SQL-подобный синтаксис

Кто использует SQL?

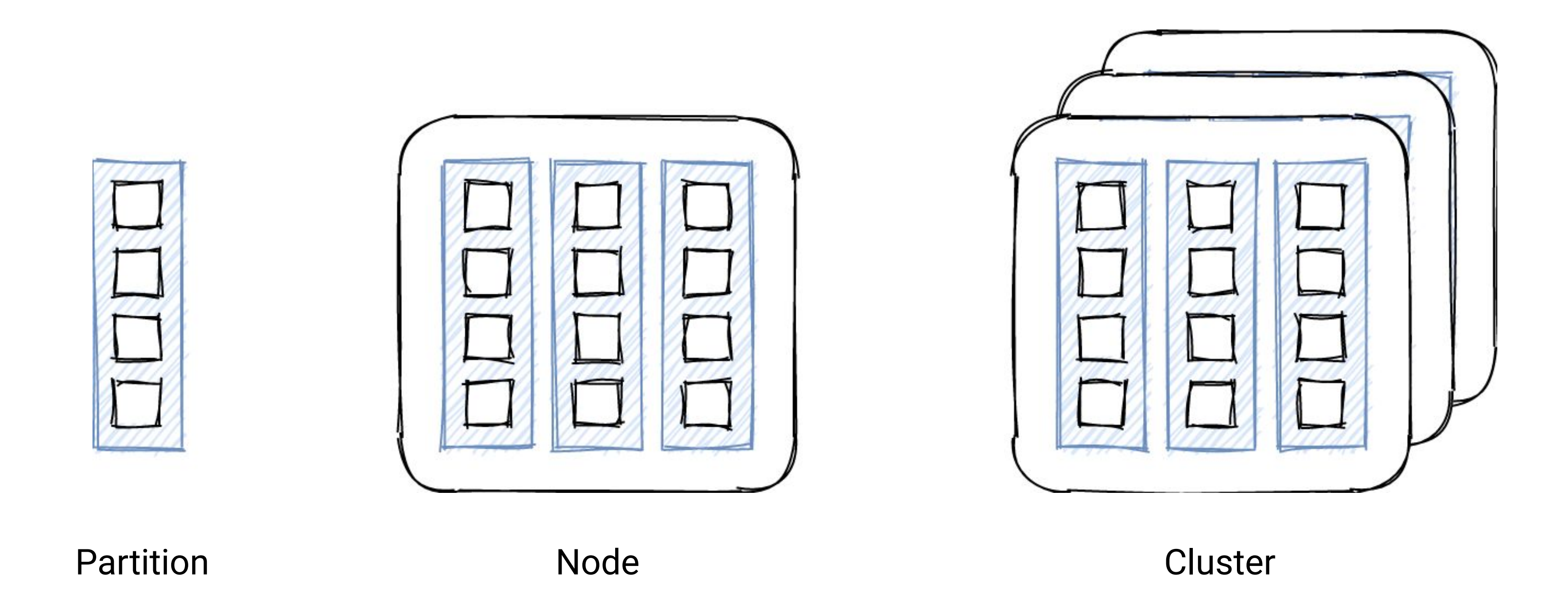
- “Классические” СУБД (MySQL, Postgres, SQL Server, Oracle)
- “Новые” системы управления данными
  - Реляционные (CockroachDB, TiDB, YugaByte)
  - BigData/Analytics (Hive, Snowflake, Dremio, Clickhouse, Presto)
  - NoSQL (DataStax*, Couchbase*)
  - Compute/streaming (Spark, ksqlDB, Apache Flink)
  - In-memory (Apache Ignite, Gigaspaces)
- Бунтари
  - MongoDB
  - Redis



* Используют SQL-подобный синтаксис

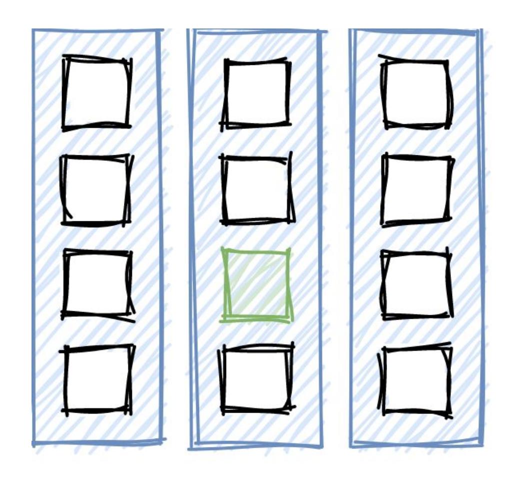
https://insights.stackoverflow.com/survey/2020

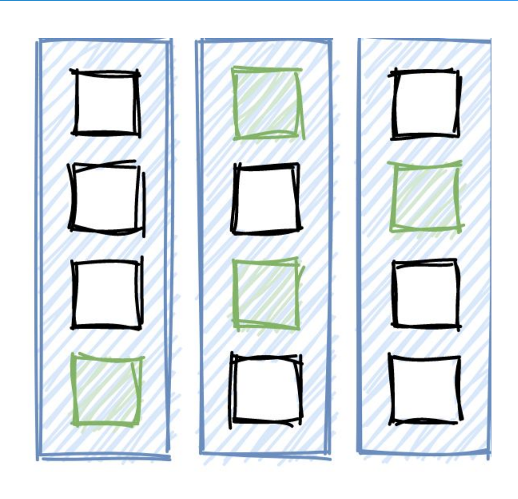
Что такое Hazelcast IMDG?



- Hazelcast IMDG — это распределенное key-value-хранилище
- Пары ключ-значение организованы в шарды (partition), которые распределены по узлам кластера

Зачем нам нужен SQL?





- Доступ только по ключу
- Хотим делать более сложный анализ: поиск по предикату, агрегации, joins, …
- Хотим интеграцию с другими продуктами через драйвера (напр., JDBC)

Что у нас было?

- Predicate API — возможность найти данные по предикату
- In-memory-индексы для ускорения поиска по предикату
  - Heap: конкурентный skip-list
  - Offheap: самописное однопоточное красно-чёрное дерево

```
IMap<Long, Person> map = …
...
map.put(1L, new Person("John"));
...
Predicate predicate = Predicates.equals("name", "John");
Collection<Person> persons = map.values(predicate);
```

Что не так с Predicate API?

- Работает с множествами, а не курсорами, может упасть с OOME
- Ограниченный функционал (например, не умеет делать join)
- Не является декларативным: необходимо компилировать и деплоить
- Необходимо разбираться с документацией

```
IMap<Long, Person> map = …
...
map.put(1L, new Person(“John”));
...
Predicate predicate = Predicates.equals(“name”, “John”);
Collection<Person> persons = map.values(predicate);
```

Что мы сделали?

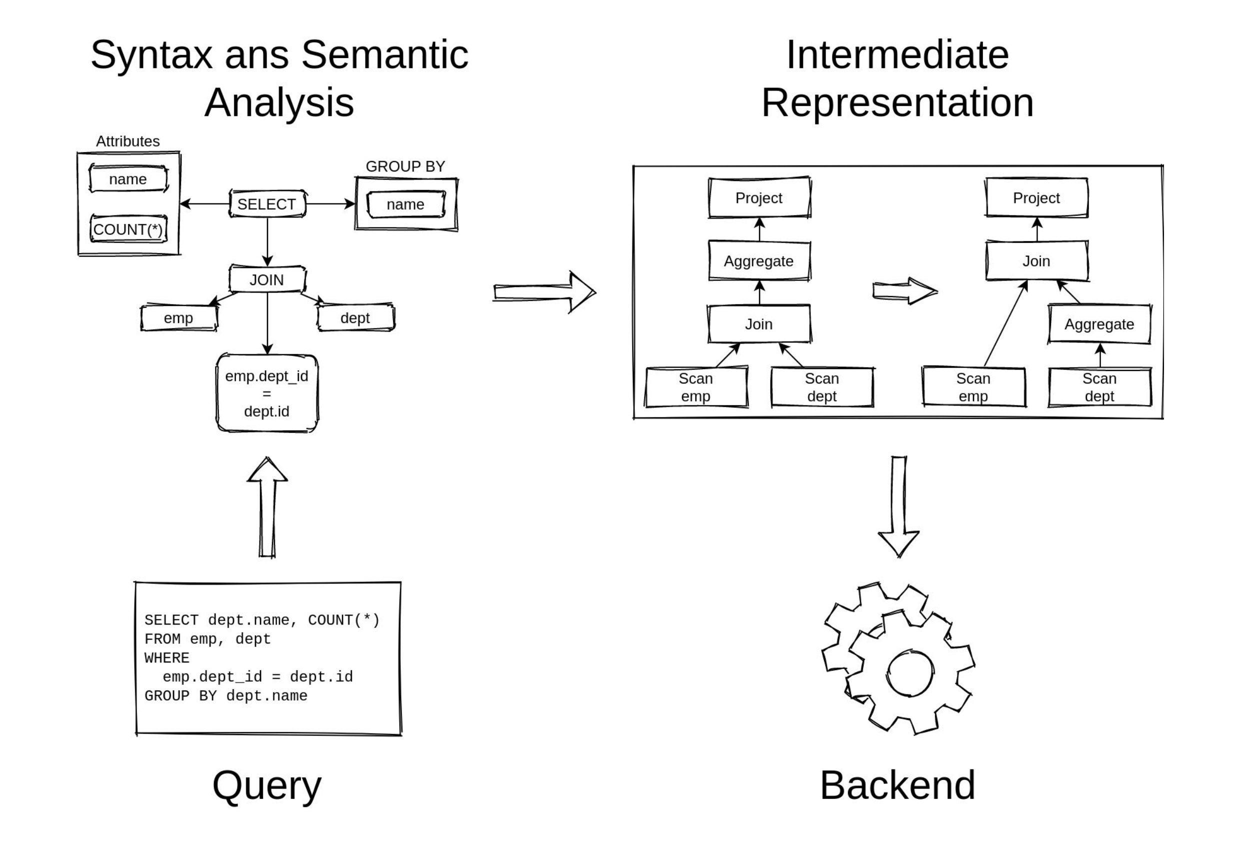
- SQL-оптимизатор на основе Apache Calcite
- Протокол распределенного выполнения запросов
- Неблокирующая кооперативная модель параллельной обработки запросов

```
IMap<Long, Person> map = instance.getMap("person");

...

try (SqlResult result = instance.getSql().query("SELECT id, name FROM person") {
    for (SqlRow row : result) {
        System.out.println("Name: " + row.getObject("name"));
    }
}
```

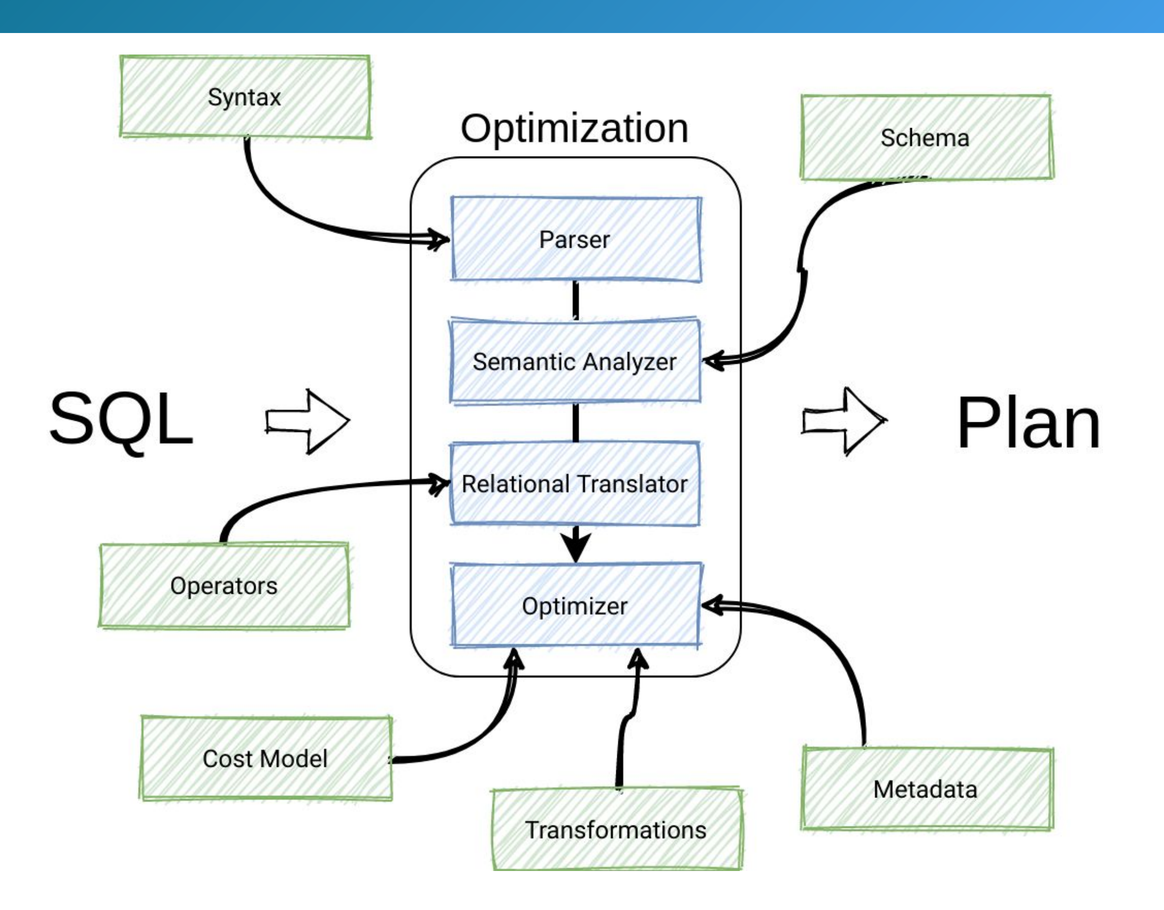
Устройство SQL-оптимизатора



Как сделать SQL-оптимизатор?

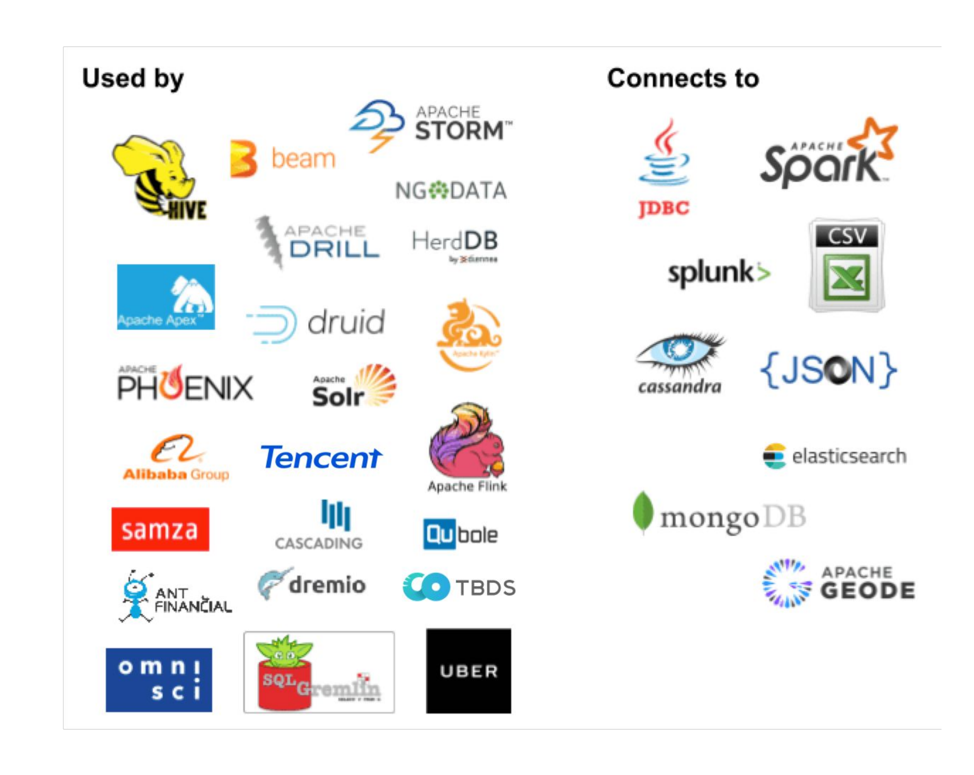
- Задача: принять строку, отдать план выполнения
- Можно писать самому:
  - Парсер
  - Семантический анализатор
  - Трансформации
- Можно собрать из готовых компонентов:
  - Генераторы парсеров или готовые парсеры
  - Apache Calcite — фреймворк для построения SQL-движков

Оптимизация с Apache Calcite



Проекты, которые используют Apache Calcite

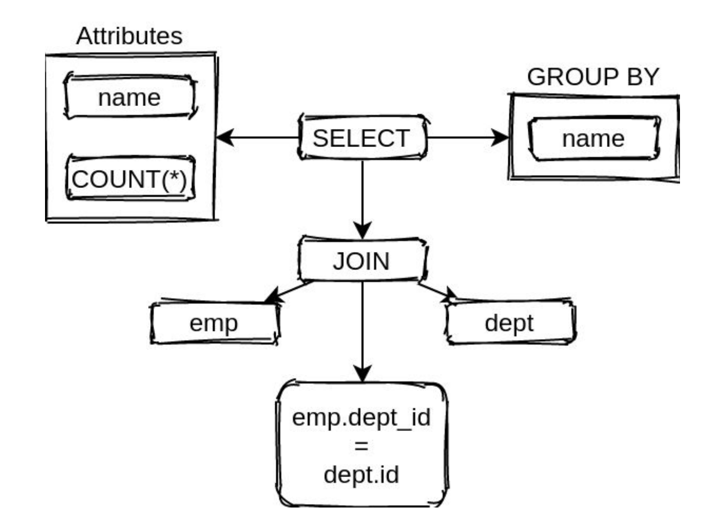
- Data Management:
  - Apache Hive
  - Apache Flink
  - Dremio
  - VoltDB
  - IMDGs (Apache Ignite, Hazelcast, Gigaspaces)
  - …
- Applied:
  - Alibaba / Ant Group
  - Uber
  - LinkedIn
  - …



https://calcite.apache.org/docs/powered_by.html

Парсинг

- Задача: превратить строку в синтаксическое дерево
- Парсинг с Apache Calcite
  - Использует JavaCC для генерации парсера
  - Имеет готовый к использованию сгенерированный парсер, SQL’03
  - Допускает расширения синтаксиса



Семантический анализ

- Задача: убедиться в семантической корректности синтаксического дерева:
  - Разрешение объектов СУБД (таблицы, колонки, …)
  - Разрешение функций
  - Вывод типов
  - Проверка реляционной семантики
- Семантический анализ в Apache Calcite
  - Предоставьте схему
  - (опционально) Предоставьте кастомные функции
  - Запустите встроенный валидатор

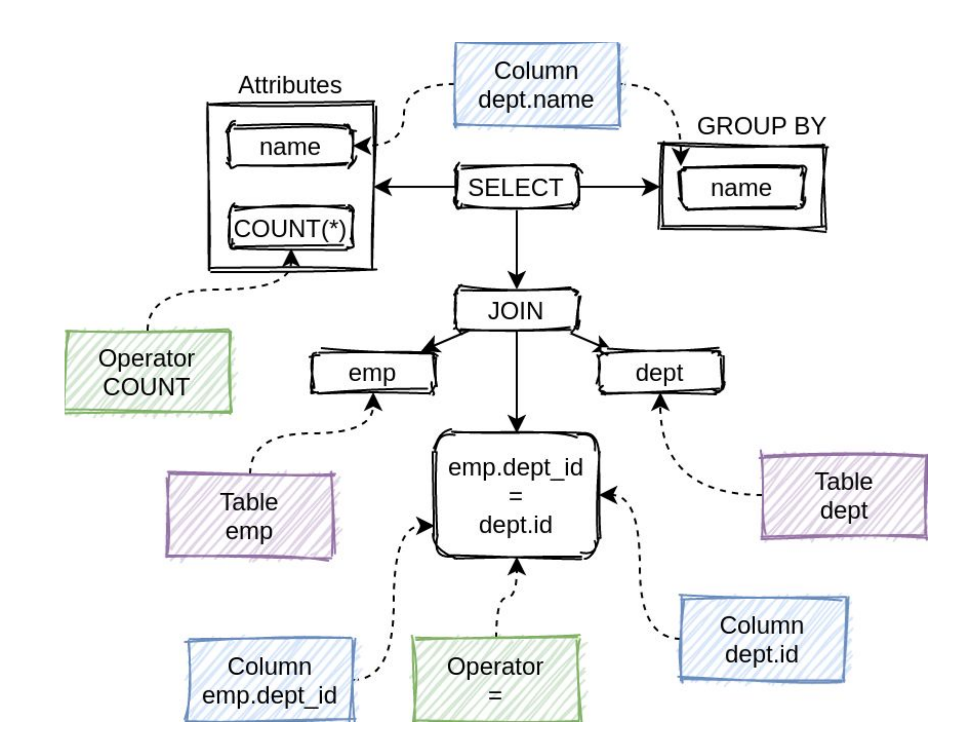
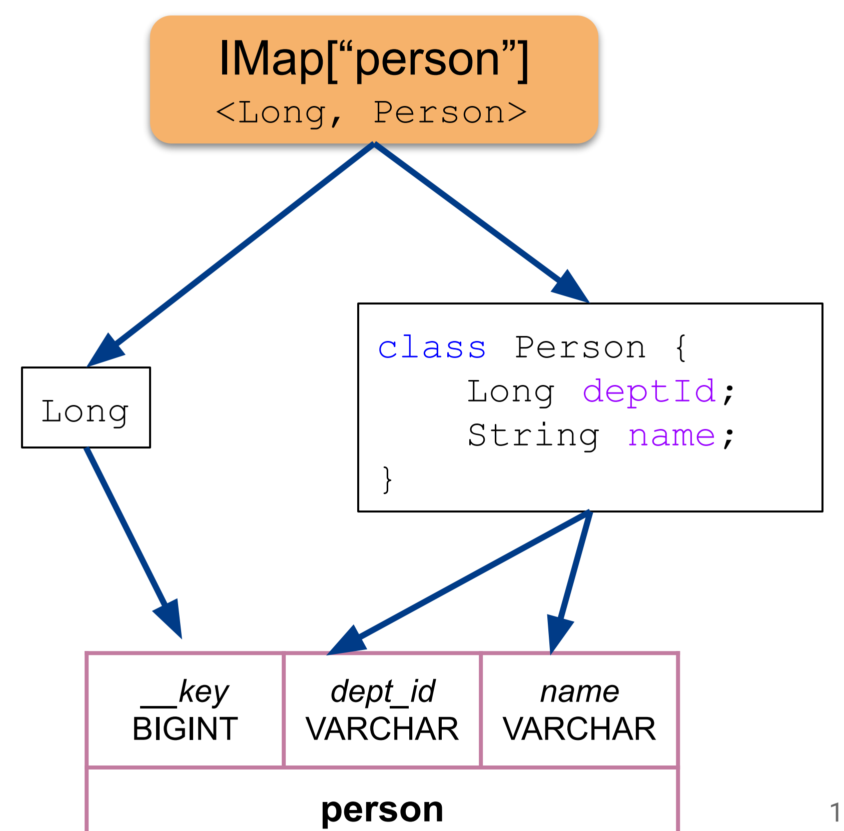
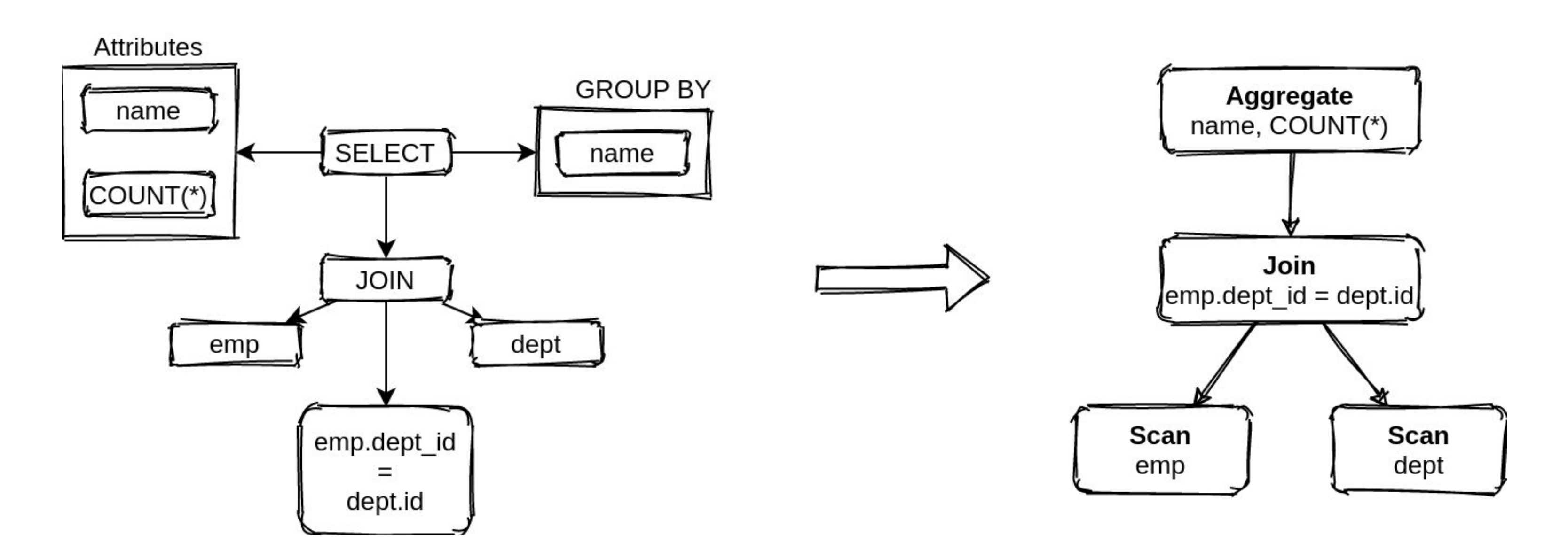


Схема в Hazelcast

- Hazelcast не имеет схемы, значит нам надо ее “придумать”
- Представляем каждый IMap как таблицу
- Находим первую key-value-пару в IMap, извлекаем из нее атрибуты интроспекцией (например, reflection)
- Во время исполнения запроса: если последующие пары имеют другие типы, бросаем ошибку



Трансляция в реляционное дерево



- Оптимизировать синтаксическое дерево проблематично из-за высокой сложности SQL-синтаксиса
- Для задачи оптимизации, удобнее представить запрос в виде дерева реляционных операторов с простой и строго ограниченной семантикой
- Большинство трансформаций Apache Calcite работают с реляционными операторами

Реляционные операторы

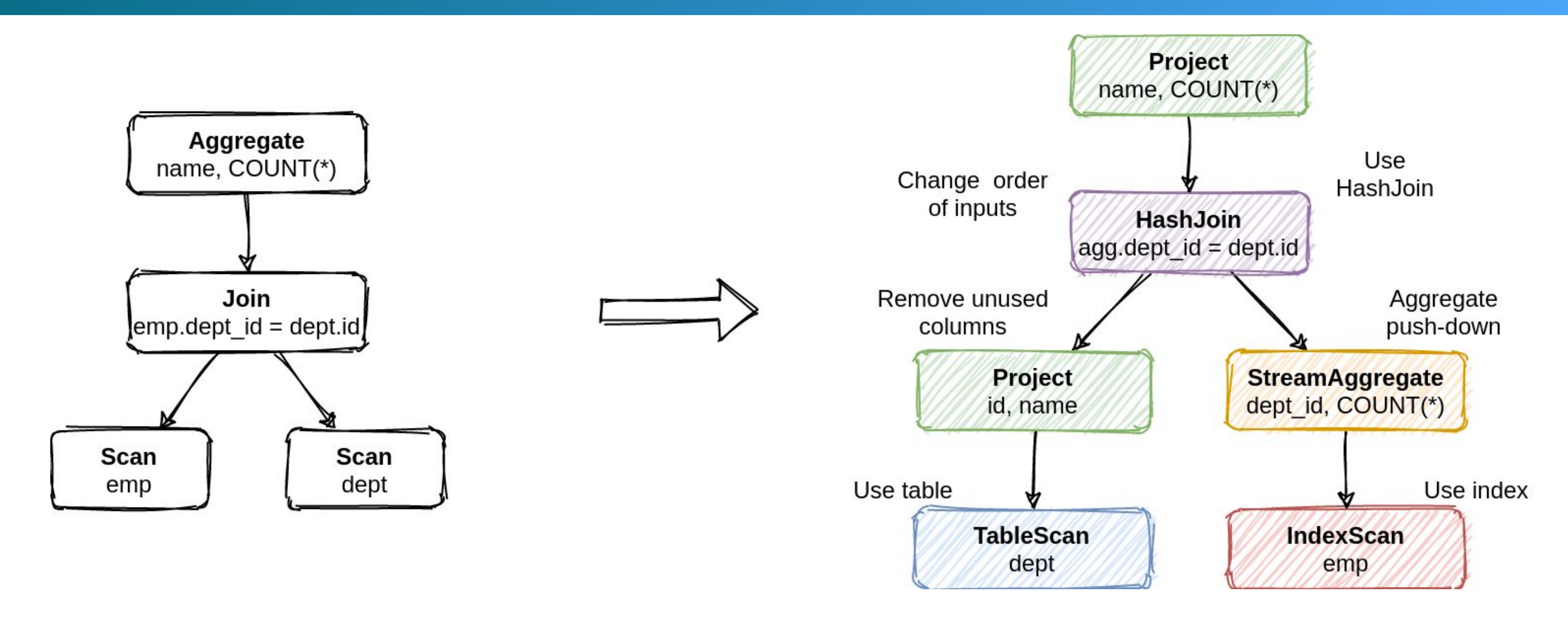
Оператор	**Описание**
Scan	Сканировать абстрактный источник данных
Project	Трансформировать кортеж (напр., `a+b`)
Filter	Отфильтровать кортежи (WHERE, HAVING)
Sort	ORDER BY / LIMIT / OFFSET
Aggregate	Агрегация
Window	Оконная агрегация
Join	Классический join двух операторов
Union/Minus/Intersect	Set-операторы

Логические и физические операторы

Логический оператор	Физические операторы
Scan	Table Scan, Index Scan
Aggregate	Hash Aggregate, Streaming Aggregate
Join	Hash Join, Merge Join, Nested Loop Join

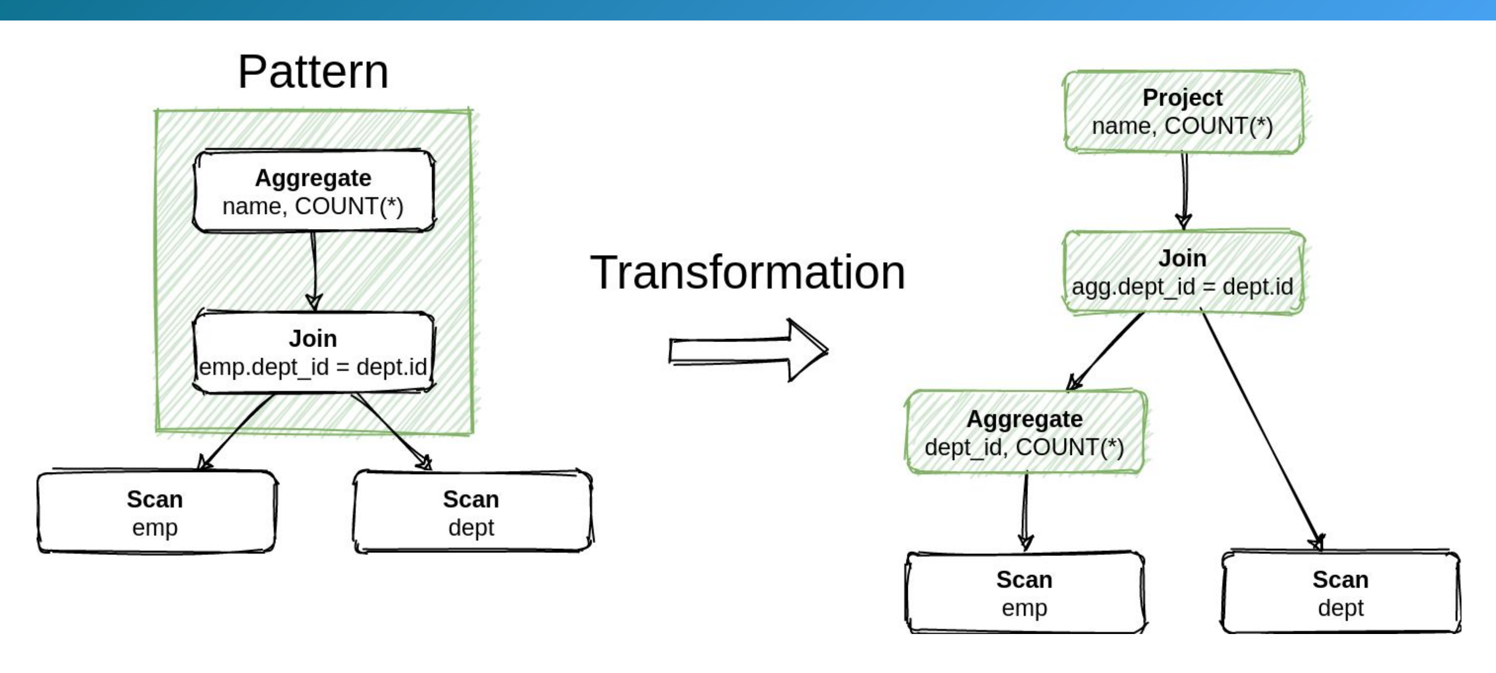
- Логические операторы – “что делать?”
- Физические операторы – “как делать?”
- Одному логическому оператору соответствует один или несколько физических

Трансформации



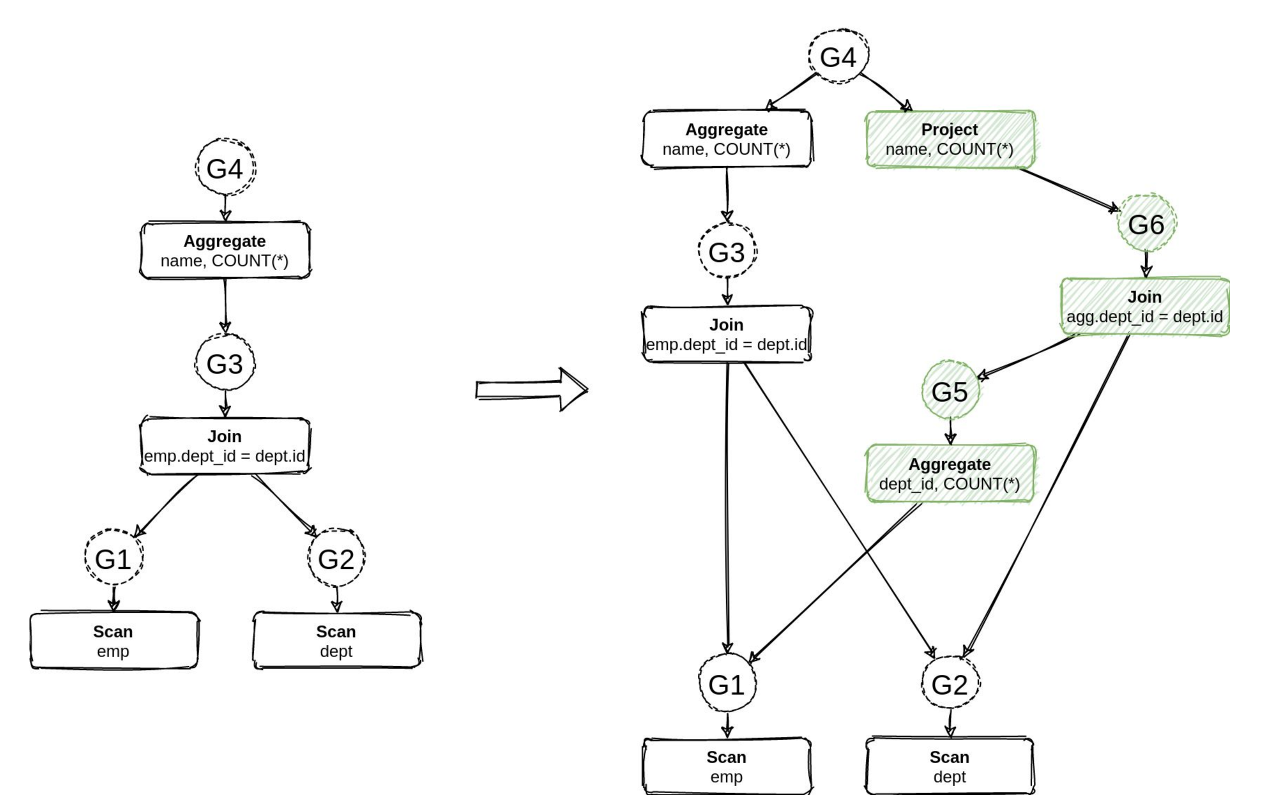
- Каждый запрос может быть выполнен несколькими способами
- Необходимо применить набор трансформаций, чтобы найти оптимальный план

Правила трансформации



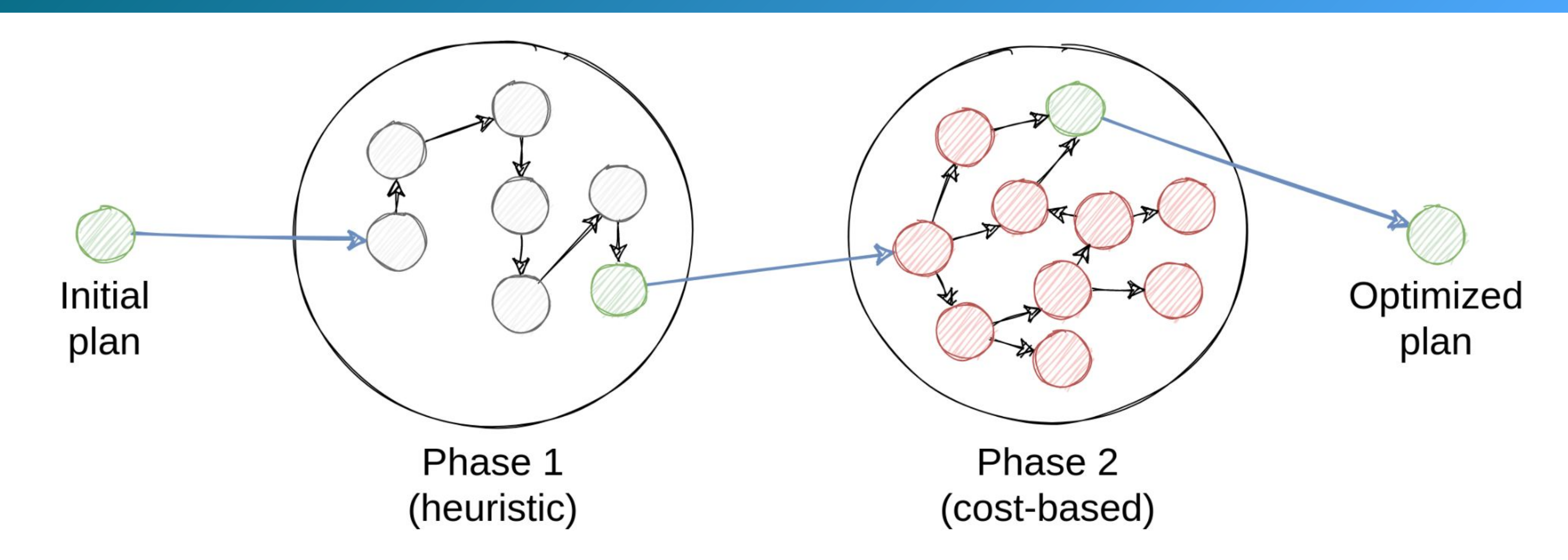
- Правило состоит из паттерна и логики трансформации
- Apache Calcite содержит порядка 100 готовых логических правил
- Но Apache Calcite ничего не знает об особенностях вашей системы, поэтому вы должны определить физические операторы и соответствующие им правила

Cost-based оптимизация



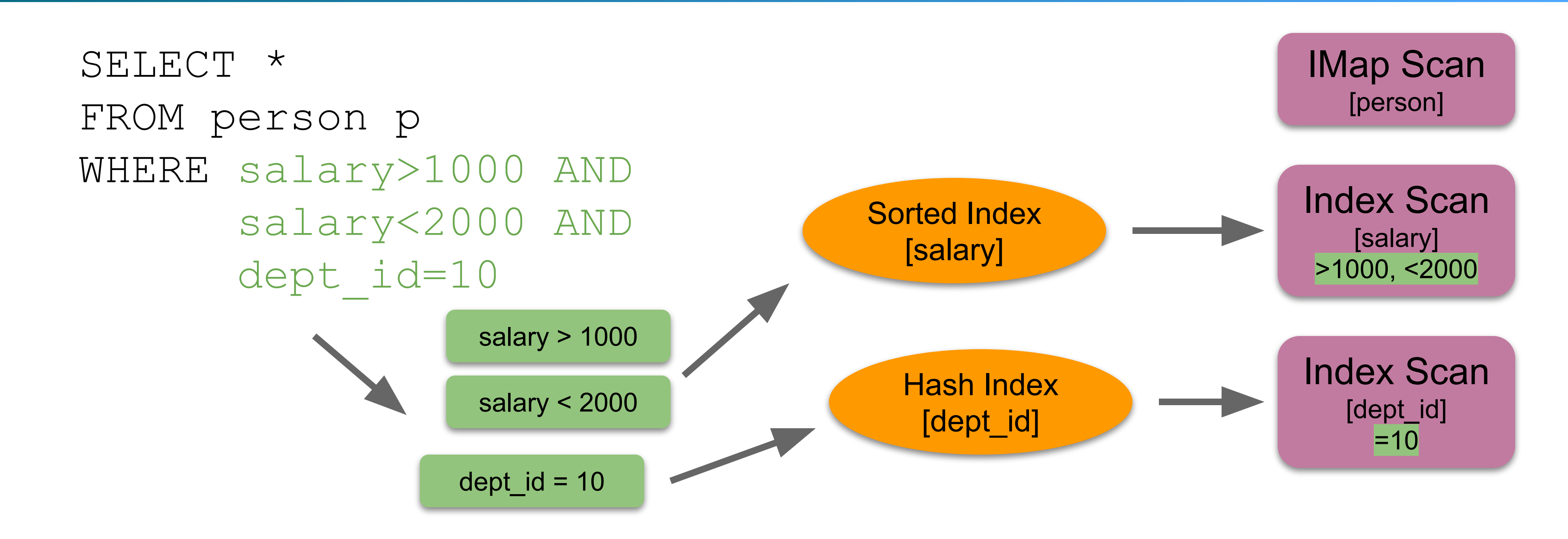
- Кодируем множество планов **одновременно** в специальной структуре данных (MEMO)

Многофазная оптимизация



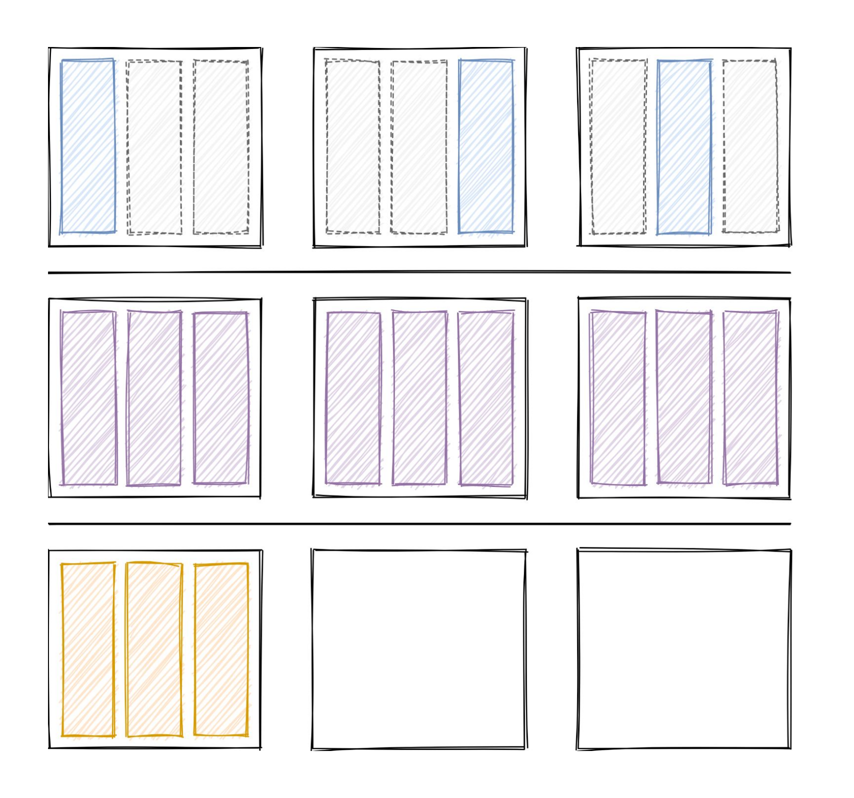
- Количество альтернативных планов может быть очень большим
- Оптимизаторы зачастую разбивают оптимизацию на несколько последовательных фаз, чтобы уменьшить количество рассматриваемых планов. Оптимальность не может быть гарантирована

Физическая оптимизация: выбор индекса



- Разбиваем конъюнктивный предикат на составные части
- Находим подходящие индексы

Физическая оптимизация: distribution



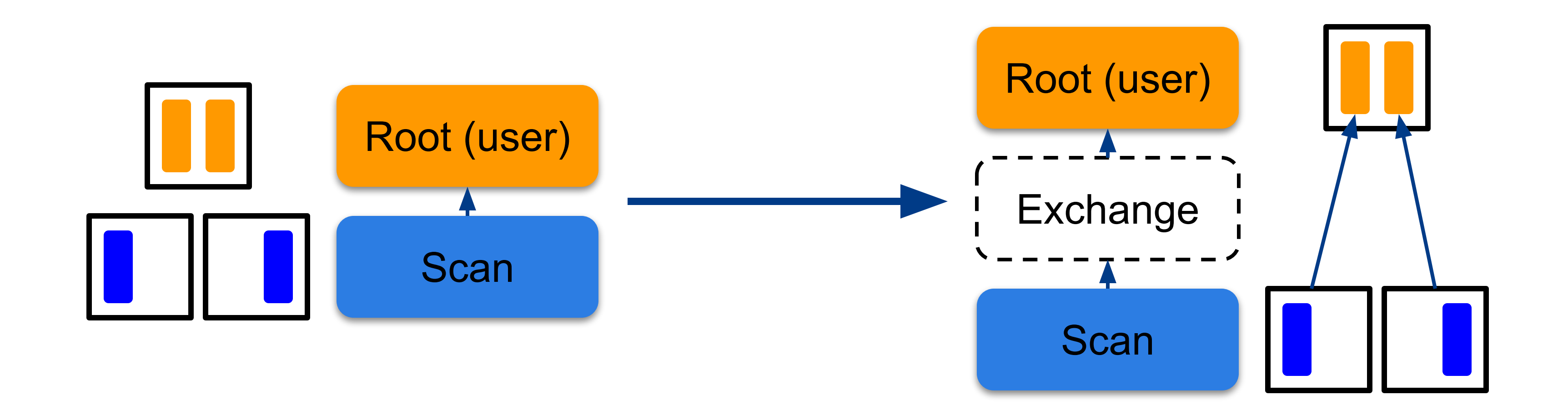
PARTITIONED

REPLICATED

SINGLETON

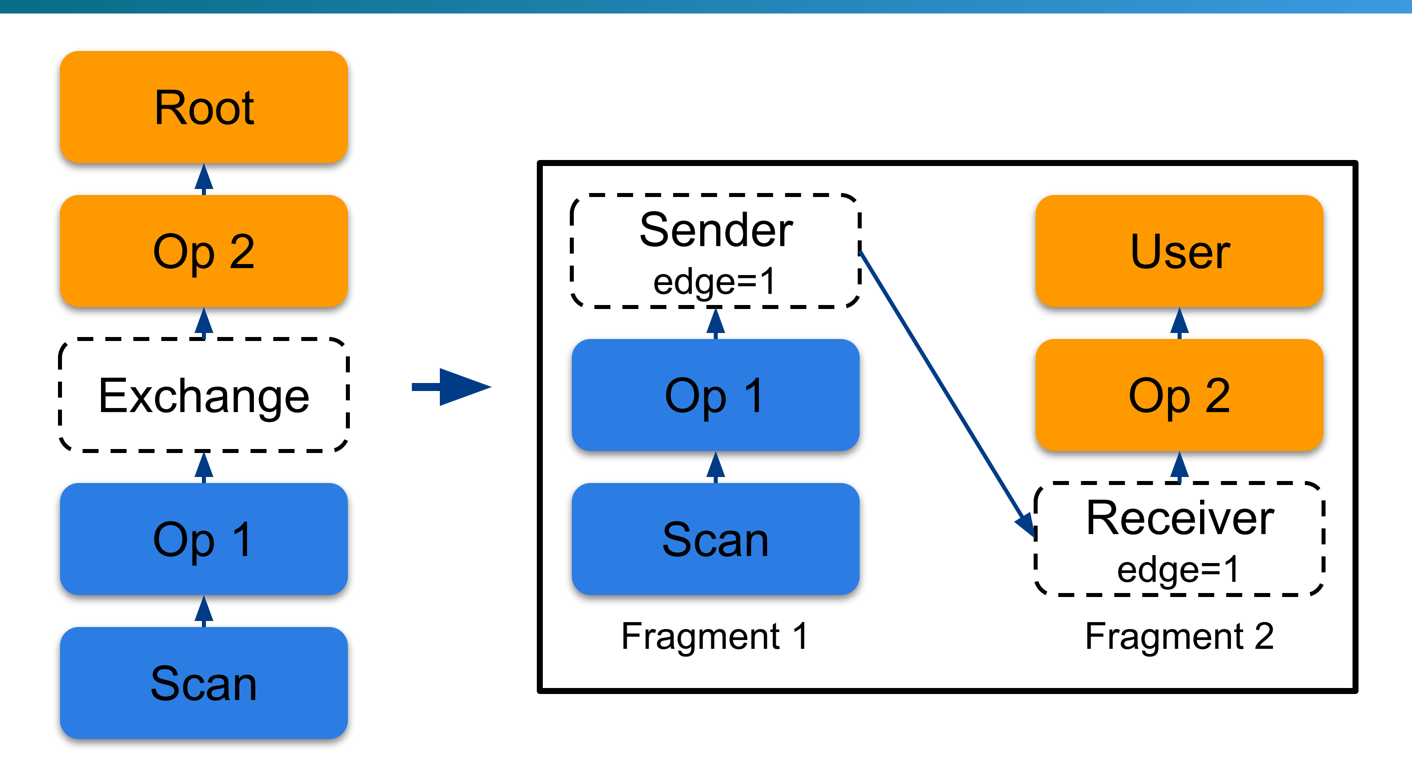
- Каждый оператор имеет свойство **distribution**, которое описывает распределение данных в кластере
- **PARTITIONED** – данные распределены по кластеру, каждый tuple в одном экземпляре (например, IMap)
- **REPLICATED** – полная копия данных на всех узлах (например, справочник)
- **SINGLETON** – данные находятся на одном узле (например, пользовательский курсор)

Физическая оптимизация: distribution



- Каждый оператор может потребовать конкретное распределение у своего инпута
- Если распределение инпута несовместимо с запрошенным, то автоматически вставляем специальный физический оператор Exchange, который моделирует перемещение данных в кластере
- Таким образом можно смоделировать выполнение **произвольного** распределенного SQL-запроса

Фрагменты

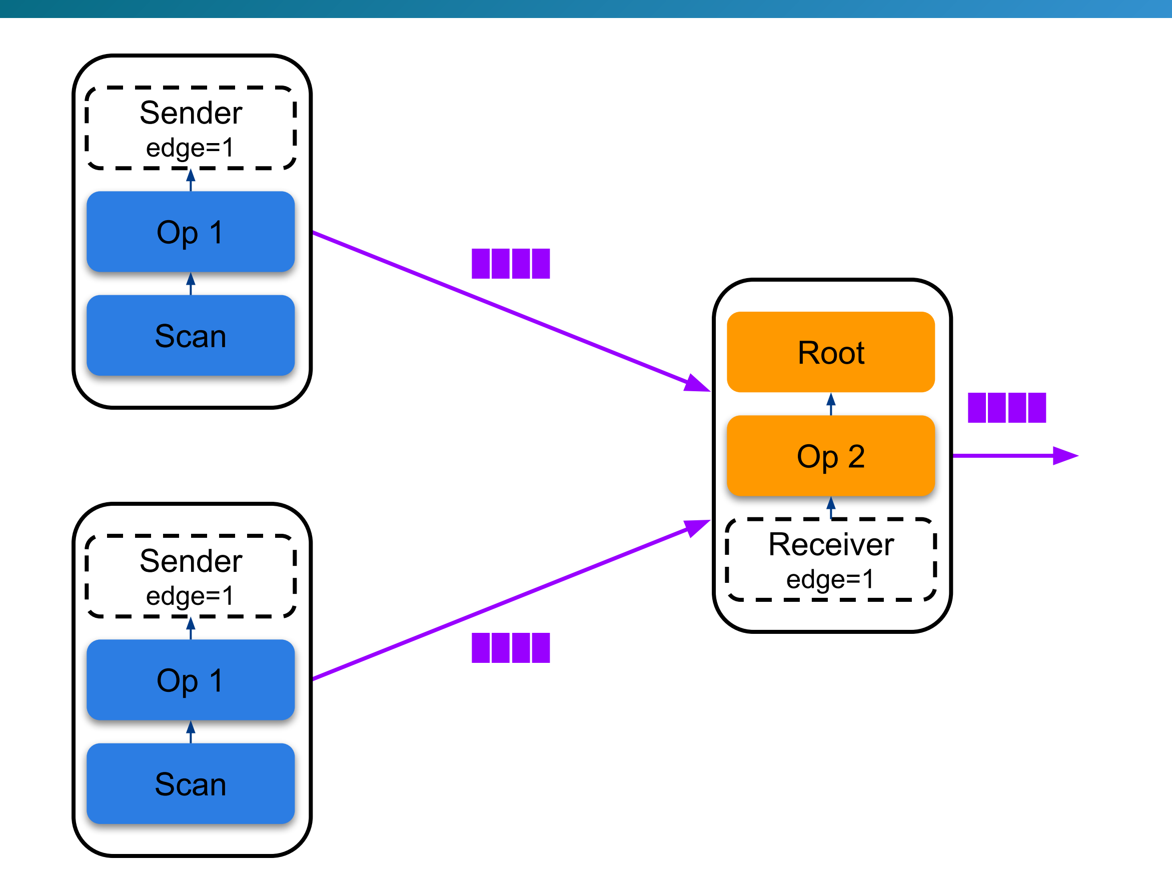


Sender – оператор, который отправляет данные

Receiver – оператор, который принимает данные

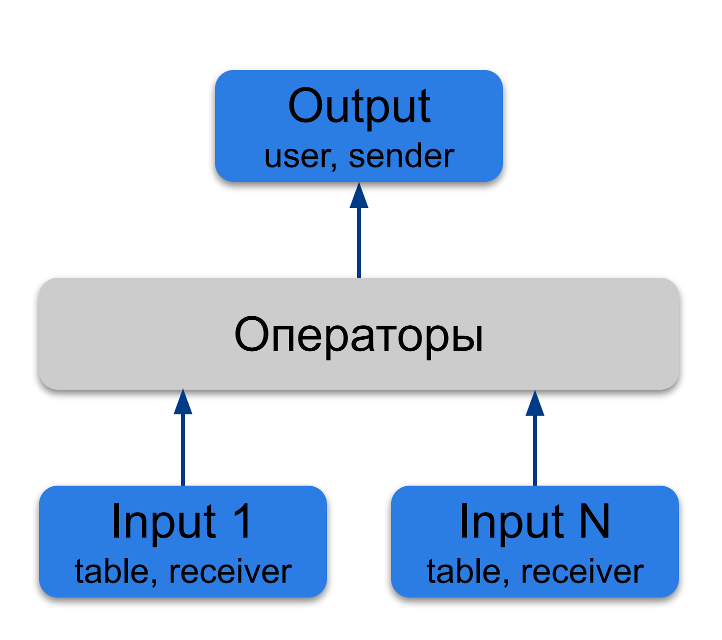
Fragment – дерево операторов, которое может быть исполнено на одном узле, независимо от других узлов и фрагментов

Фрагменты



- Каждый фрагмент запускается на одном или нескольких узлах
- Данные передаются между связанными парами sender-receiver, пока не достигнут пользователя
- Нет промежуточных материализаций
  - Высокая производительность
  - Отсутствие fault-tolerance: в случае падения узла, запрос необходимо перезапускать

Структура фрагмента

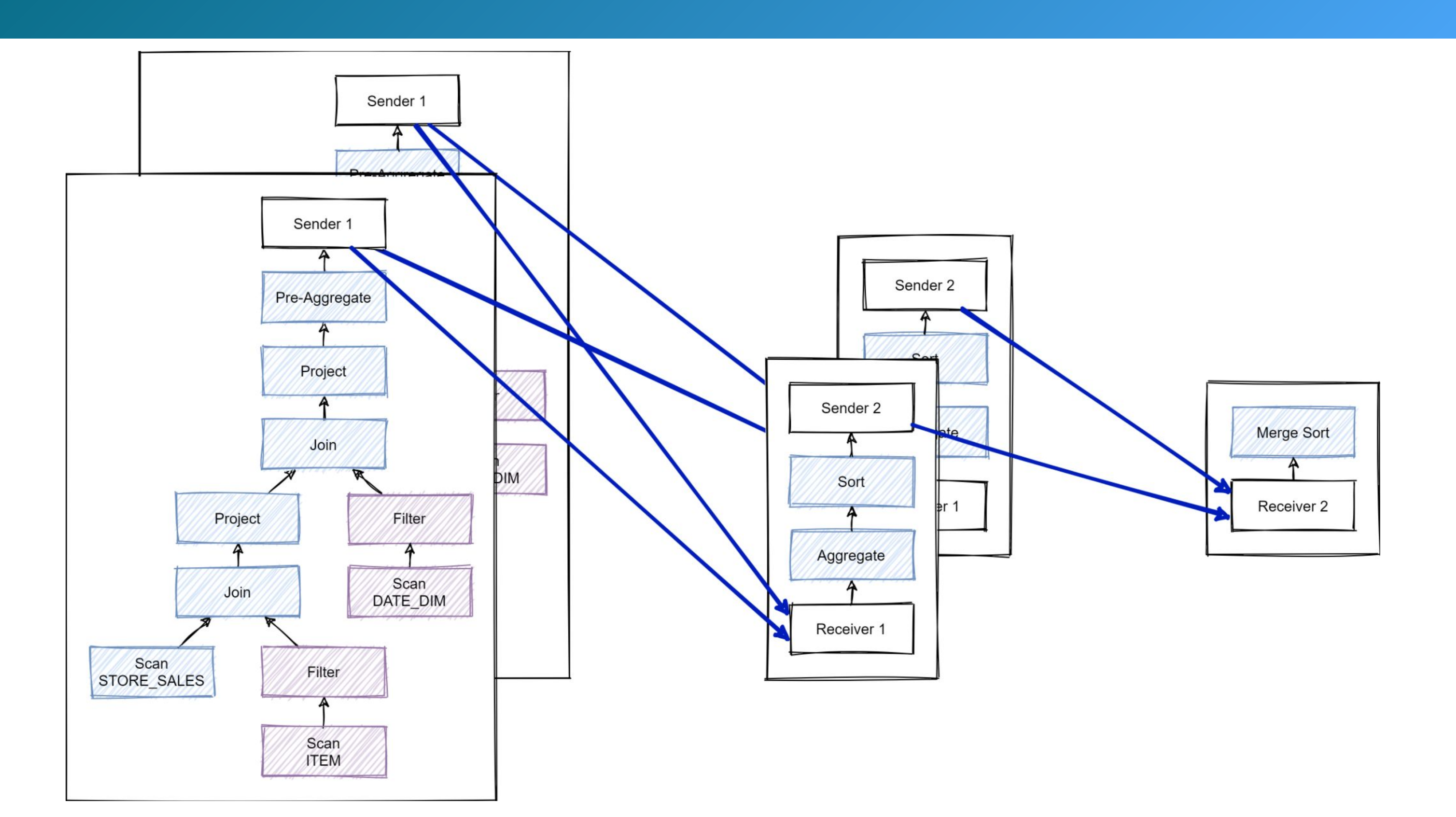


- Один или несколько input'ов (scan, receiver)
- Один output (sender, user)
- Произвольное количество промежуточных операторов

Пример: TPC-DS Q3

```
select dt.d_year,
       item.i_brand_id brand_id,
       item.i_brand brand,
       sum(ss_ext_sales_price) sum_agg
 from date_dim dt,
      store_sales,
      item
 where dt.d_date_sk = store_sales.ss_sold_date_sk
   and store_sales.ss_item_sk = item.i_item_sk
   and item.i_manufact_id = 128
   and dt.d_moy=11
 group by dt.d_year,
      item.i_brand,
      item.i_brand_id
 order by dt.d_year,
          sum_agg desc,
          brand_id
 limit 100
```

Пример: TPC-DS Q3



Volcano Model

```
interface Exec {
    void open();
    Row next();
    void close();
}
```

120 IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 6, NO. 1, FEBRUARY 1994

Volcano—An Extensible and Parallel Query Evaluation System

Goetz Graefe

***Abstract*—To investigate the interactions of extensibility and parallelism in database query processing, we have developed a new dataflow query execution system called Volcano. The Volcano effort provides a rich environment for research and education in database systems design, heuristics for query optimization, parallel query execution, and resource allocation.**

Volcano uses a standard interface between algebra operators, allowing easy addition of new operators and operator implementations. Operations on individual items, e.g., predicates, are imported into the query processing operators using *support functions*. The semantics of support functions is not prescribed; any data type including complex objects and any operation can be realized. Thus, Volcano is *extensible* with new operators, algorithms, data types, and type-specific methods.

Volcano includes two novel *meta-operators*. The *choose-plan* meta-operator supports *dynamic query evaluation plans* that allow delaying selected optimization decisions until run-time, e.g., for embedded queries with free variables. The *exchange* meta-operator supports *intra-operator parallelism* on partitioned datasets and both *vertical and horizontal inter-operator parallelism*, translating between demand-driven dataflow within processes and data-driven dataflow between processes.

All operators, with the exception of the exchange operator, have been designed and implemented in a single-process environment, and parallelized using the exchange operator. Even operators not yet designed can be parallelized using this new operator if they use and provide the interator interface. Thus, the issues of data manipulation and parallelism have become *orthogonal*, making Volcano the first implemented query execution engine that effectively combines extensibility and parallelism.

***Index Terms*—Dynamic query evaluation plans, extensible database systems, iterators, operator model of parallelization, query execution.**

I. Introduction

IN ORDER to investigate the interactions of extensibility, efficiency, and parallelism in database query processing and to provide a testbed for databse systems research and education, we have designed and implemented a new query evaluation system called Volcano. It is intended to provide an experimental vehicle for research into query execution techniques and query optimization optimization heuristics rather than a database system ready to support applications. It is not a complete database system as it lacks features such as a user-friendly query language, a type system for instances (record definitions), a query optimizer, and catalogs. Because of this focus, Volcano is able to serve as an experimental vehicle for a multitude of purposes, all of them open-ended, which results in a combination of requirements that have not been integrated in a single system before. First, it is modular and extensible to enable future research, e.g., on algorithms, data models, resource allocation, parallel execution, load balancing, and query optimization heuristics. Thus, Volcano provides an infrastructure for experimental research rather than a final research prototype in itself. Second, it is simple in its design to allow student use and research. Modularity and simplicity are very important for this purpose because they allow students to begin working on projects without an understanding of the entire design and all its details, and they permit several concurrent student projects. Third, Volcano's design does not presume any particular data model; the only assumption is that query processing is based on transforming sets of items using parameterized operators. To achieve data model independence, the design very consistently separates set processing *control* (which is provided and inherent in the Volcano operators) from *interpretation and manipulation* of data items (which is imported into the operators, as described later). Fourth, to free algorithm design, implementation, debugging, tuning, and initial experimentation from the intricacies of parallelism but to allow experimentation with parallel query processing. Volcano can be used as a single-process or as a parallel system. Single-process query evaluation plans can already be parallelized easily on shared-memory machines and soon also on distributed-memory machines. Fifth, Volcano is realistic in its query execution paradigm to ensure that students learn how query processing is really done in commercial database products. For example, using temporary files to transfer data from one operation to the next as suggested in most textbooks has a substantial performance penalty, and is therefore used in neither real database systems nor in Volcano. Finally, Volcano's means for parallel query processing could not be based on existing models since all models explored to date have been designed with a particular data model and operator set in mind. Instead, our design goal was to make parallelism and data manipulation *orthogonal*, which means that the mechanisms for parallel query processing are independent of the operator set and semantics, and that all operators, including new

Manuscript received July 26, 1990; revised September 5, 1991. This work was supported in part by the National Science Foundation under Grants IRI-8996270, IRI-8912618, and IRI-9006348, and by the Oregon Advanced Computing Institute (OACIS).

The author is with the Computer Science Department, Portland State University, Portland, OR 97207-0751.

IEEE Log Number 9211308.

Volcano Model: пример

```
class FilterExec implements Exec {

    Exec input;
    Expression filter;

    Row next() {
        while (true) {
            Row row = input.next();

            if (filter.eval(row))
                return row;
        }
    }
}
```

Volcano Model: batching

```
interface Exec {
    void open();
    Row next();
    void close();
}
```

→

```
interface Exec {
    void open();
    List<Row> next();
    void close();
}
```

Volcano Model: неблокирующее выполнение

```
interface Exec {
    void open();
    List<Row> next();
    void close();
}
```

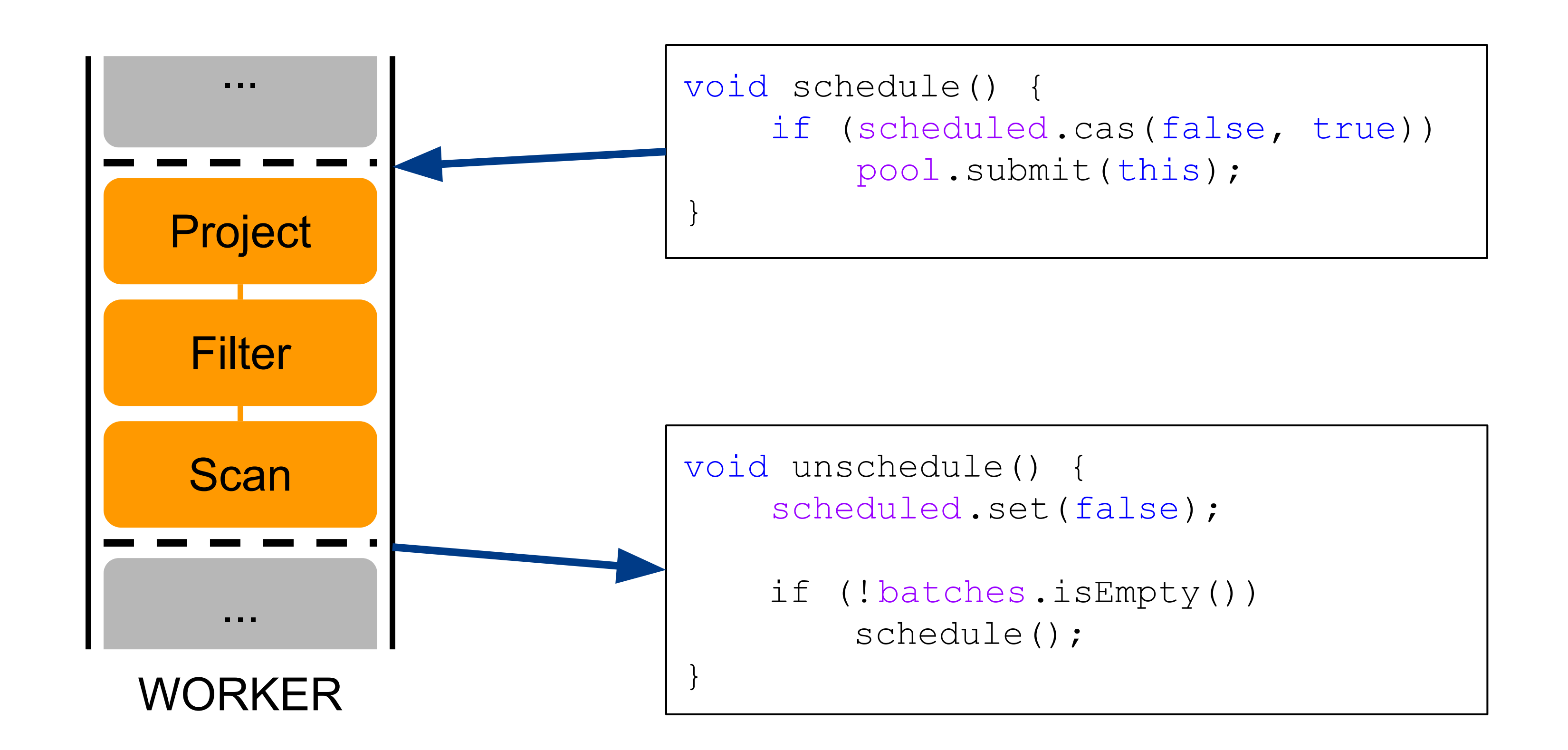


```
interface Exec {
    void open();
    IterationResult next();
    List<Row> currentBatch();
    void close();
}

enum IterationResult {
    FETCHED,
    FETCHED_DONE,
    WAIT
}
```

- Если оператор не может продолжить работу, он возвращает WAIT
- Родительские операторы передают WAIT вверх по стеку, что приводит к освобождению потока

Запуск фрагмента



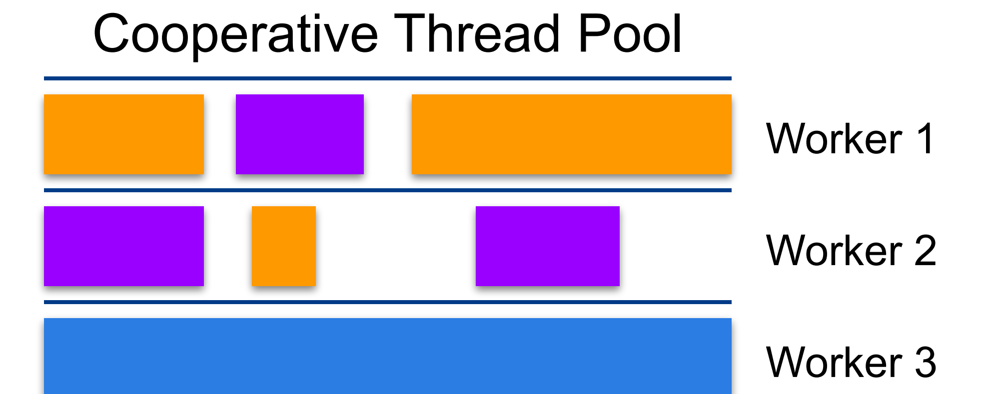
Кооперативность



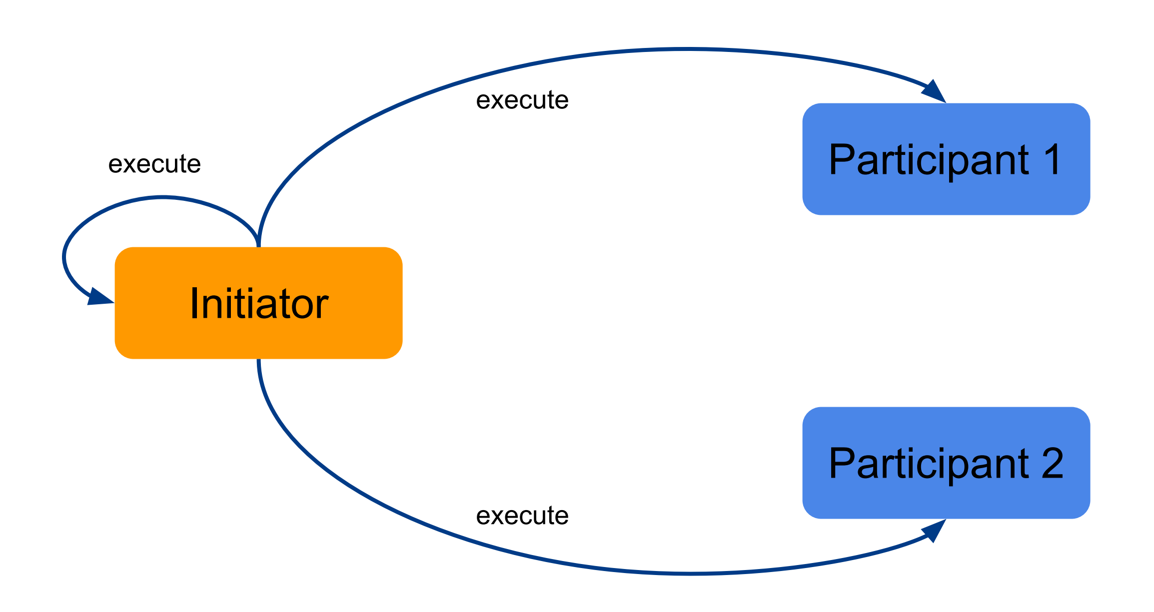
F1

F2

F3

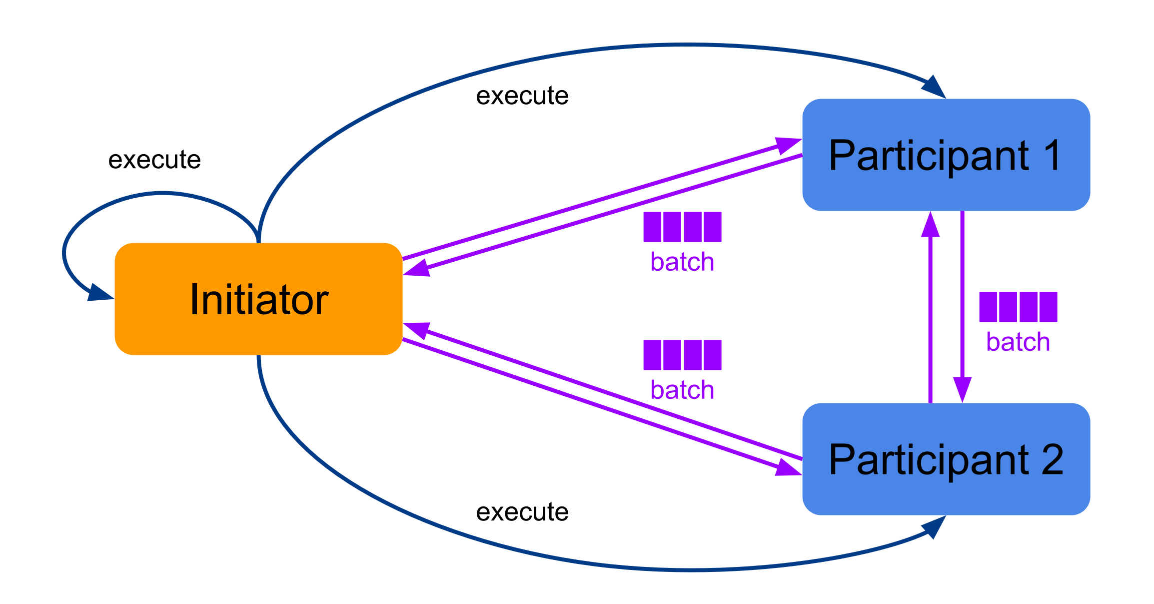


Протокол: запуск запроса



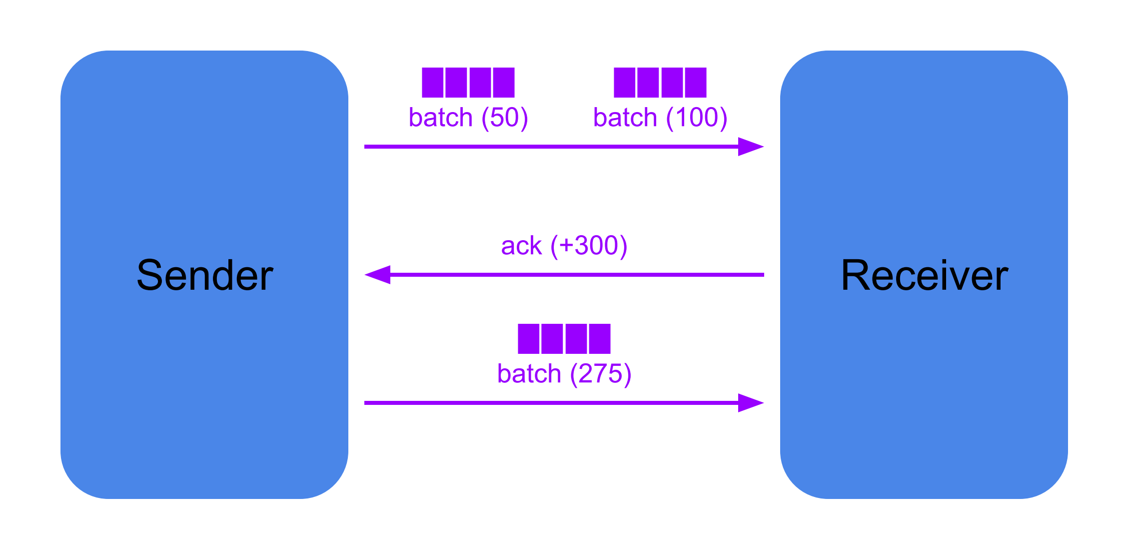
- **Initiator** — координирует выполнение запроса
- **Participant** — обычный участник
- Initiator знает всех узлов-участников, и отсылает им **execute**-запрос

Протокол: запуск запроса



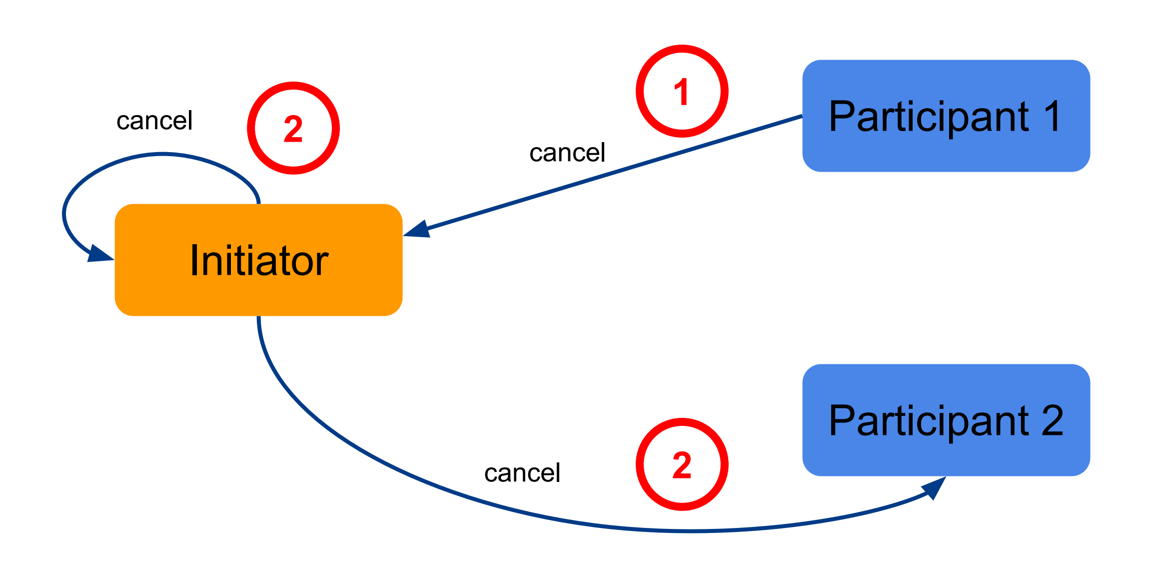
- **Initiator** — координирует выполнение запроса
- **Participant** — обычный участник
- Initiator знает всех узлов-участников, и отсылает им **execute**-запрос
- Участники начинают обмениваться **batch**-сообщениями с данными

Протокол: flow control



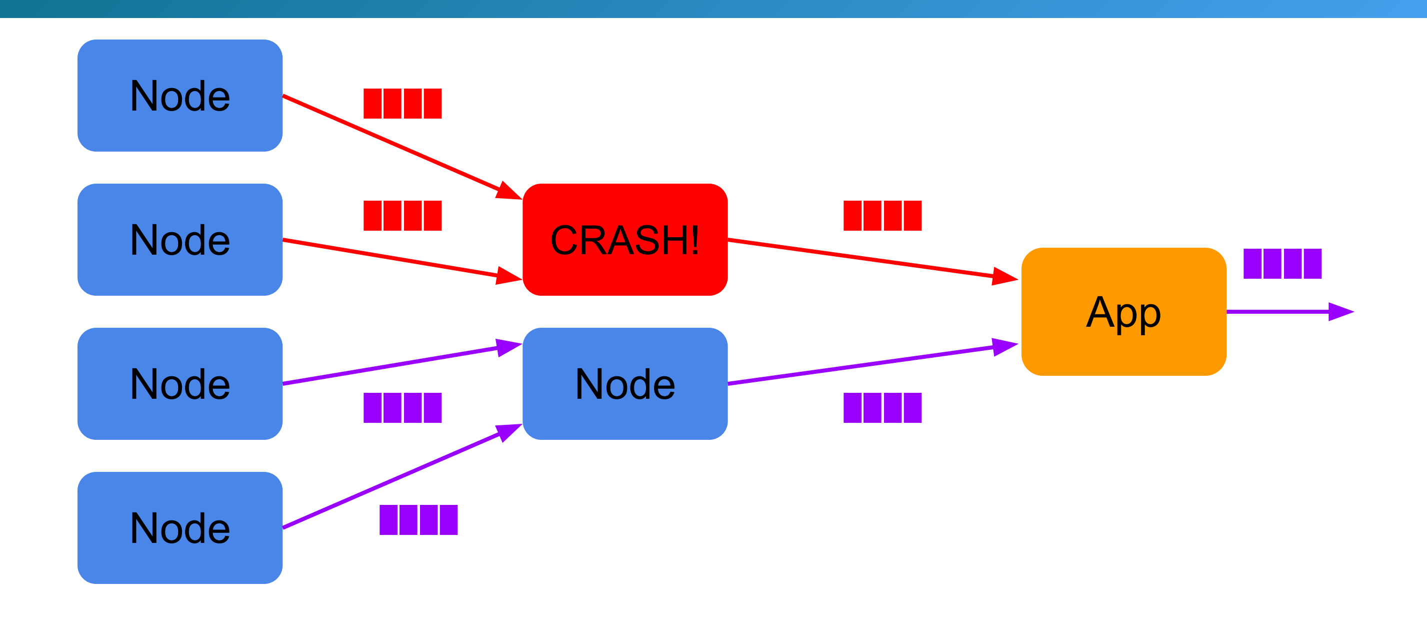
- Sender и receiver договариваются о количестве “кредитов”.
- Sender уменьшает количество “кредитов” соразмерно количеству отправленных байт. Отправка сообщений приостанавливается, когда “кредиты” исчерпаны.
- Receiver периодически отправляет **ack-**сообщение, которое увеличивает количество “кредитов” на sender’e.

Протокол: отмена запроса



- Любой узел может потребовать отменить запрос (ошибка, потеря узла, запрос пользователя).
- Participant отправляет cancel сообщение на initiator'у.
- Initiator делает broadcast сообщения на остальных participant'ов.

Протокол: отказоустойчивость



- Так как движок не материализует промежуточные результаты, продолжить выполнение при выходе узла из строя проблематично.
- Движок бросит ошибку, но никогда не вернет некорректный результат.
- Рекомендуем пользователю перезапустить запрос. Достаточно для OLTP, проблематично для OLAP.

Итого

- Оптимизатор на основе Apache Calcite
  - Встроенные оптимизации Apache Calcite
  - Выбор вторичных индексов
  - Планирование перемещения данных в кластере (“exchange”)
- Runtime
  - DAG из фрагментов, которым можно описать запросы произвольной сложности
  - Неблокирующие Volcano-style-итераторы с буферизацией
  - Кооперативная многозадачность
- Сетевой протокол
  - Оптимизирован под минимизацию latency
  - Flow control для избежания перегрузки узлов
  - Задача отказоустойчивости переложена на пользователя

Links

- Спикер
  - https://www.linkedin.com/in/devozerov/
  - https://twitter.com/devozerov
- Hazelcast SQL:
  - https://github.com/hazelcast/hazelcast/tree/master/hazelcast-sql
- Apache Calcite:
  - https://calcite.apache.org/
  - https://github.com/apache/calcite
- Блог Querify Labs:
  - https://www.querifylabs.com/blog